Peter Burkinshaw

COMAL Standard Extension proposals

It is often useful to be able to handle part of an array as a unit for purposes such as initialisation or passing to a procedure as a single parameter. For initialisation, it is also useful to be able to assign the same value to all element of a numerical array. These proposals describe extensions implemented in Teli COMAL release 2.1.

1. It is proposed to allow an 'alias' for part of an array to be created. The 'aliased' part must consist of contiguous elements without gaps. For programs using this feature to be portable, assumptions are made about the way arrays are stored. In Teli (and RC) COMAL they are stored in an order in which the 'rightmost' index varies from one stored element to the next. The 'leftmost' index varies least often (do any implementations differ?).

   The proposed syntax is exemplified by:

```
10 DIM a(100) // declare base array
20 DIM a(11): DIM b(10) ,c(20)
```

   The second DIM statement means "Array b, with 10 elements, starts at a(11), and ends at a(20)". Any extra arrays such as c will be allocated to the next free place, in this case a(21) etc.

   Notes

   a. Array a must already exist before it can be redefined.
   b. The redefined portion must lie entirely within the base array.
   c. Array b, c are not initialised, it takes its initial values from the existing base array values.
   d. Array b, c need not have the same number of dimensions as array a, or as each other.
   e. Array b need not have the same type as array a, e.g. a may be real, and b# could be integer, however programs using this will not in general be portable. (In Teli COMAL the base array must be a numeric type.)
   f. More than one alias may be defined for an array, and aliases may be defined using an existing alias as a base array, however all aliases on one statement use the same base address given by the first entry.

2. It is proposed that an entire array may be assigned a value as a single statement

E.g.

```
10 DIM a(100)
20 a=1 // (or expression)
```

...assigns the real value 1 to every element of a.
Some languages use the syntax

```
a() = 1
```

but as COMAL does not permit the use of the same name as a scalar and array, the extra qualification is unnecessary, and is in fact more likely to lead to errors through inadvertent omission of an index, whereas one is hardly likely to omit the entire parenthetical part other than by using the wrong name.

Notes

a. If a is an alias for a part of a then only those elements of b that comprise a will be assigned the value.
b. The value assigned to the array is always the same. thus a = RND assigns the same random number to all array elements.
c. The assigned value may not be an array, that is, you may not write a = b where a and b are both arrays (the copying of entire arrays can be speeded up by creating a one dimensional alias for each, and a single level FOR loop can then be used to copy the values, regardless of the dimensionality of the original arrays).

3. It is proposed that an entire array may be read, written, input or printed as a single statement

   E.g.

```
10 DIM a(10,10)
20 a:=1
30 PRINT a;
```

   The entire array starting with a(1,1) .. a(1,2).. a(10,10) is printed using the spacing defined by the punctuation mark following array name, and the current ZONE and MARGIN. PRINT USING may also be used to refine the format.

4. It is proposed to allow a single level repeat factor to be used in USING strings.

   E.g.

```
10 DIM a(10,10)
20 a:=1
30 PRING USING "10(####)/":a;
```

   The syntax is that a decimal number 1..99 followed immediately by (defines a repeat factor. The repeated field terminates with the first following), that is not enclosed in apostrophes as part of a literal string.

5. Multiple assignment

   When initialising different variables to the same value, it is tedious to write each as a separate assignment. Many BASIC's offer the capability of abbreviating this process by writing a list of variable names with some separator up to the final assigned variable.

   It is proposed to allow a list of variables, of the same type, e.g. all reals, all integers, or all strings to be assigned by writing a list separated by commas, thus
   a,b,c,d := 1
   array elements would be allowed as would entire arrays of the same type.

6. Whilst implementing the negative substring specifiers, whereby the substring index -1 means "the last character of .." It 'fell-out' that substring specifier 0 computed the address of the character one beyond the end of the string. As this address is legal for assignment it appears to be a very convenient way to build up a string piecemeal without the need to maintain a running index of the current length of the string.

E.g.
```
10 b = 1000
20 DIM a$ OF b
30 LOOP b
40 a$(0:) = "*"
50 ENDLOOP
```

Unless you have the equivalent string function to the BASIC STRING$ ("*", 1000), this is probably the quickest, and most economical way of initialising a string. If you have a short form LOOP statement, lines 30 - 50 can be combined.

As a further refinement, you may consider that a$(0:) "opens" the end of the string, so that one can write.

```
a$(0:) = "***"
```

which is sematically equivalent to

```
a$:= a$ + "***"
```

.. or

```
a$:+"***"
```

in some COMALs.

7. Print separators

It is proposed to add colon : to the list of print separators. When a colon is reached in the print list, a <new-line> is generated. This allows several lines to be printed using one PRINT statement, and considerably reduces the need to write empty PRINT statements.

As a further extension, it is proposed that more than one separator may be used between items in the print list. Thus PRINT a:,;b would mean "Print 'a' followed by a new line, then skip to the next zone and print a space followed by 'b'". It should also be allowed to start a print list with separators. The only restriction that seems reasonable is that a print list may not consist entirely of separators.

8. Compound statements

It is proposed to introduce the possibility to combine a number of procedure calls and assignments into one statement, as an extension. The syntax would be

<compound statement> ::=[procedure call statement;]
<assignment statement> ;<procedure call statement>

<procedure call statement>:: =[<assignment statement>]
;<compound statement>

9. MAXIMUM (<par> ,<par> ..)

returns the maximum value in the list of parameters.
<par> may be one of
a) numeric expression
b) real or integer array name

1. If every element of the list is of type integer, then the result will be of type integer, otherwise it will be of type real

2. The parameter list may contain any number of items, but at least one.

10. MINIMUM (<par> ,<par>..)

returns the minimum value in the list of parameters.
In all other respects, the same comments apply as for MAXIMUM.

11. SIGMA (<numeric-array> ,index)

returns the sum of the elements of the array, each element being raised to the power <index>. If <index> is omitted, 1 is assumed.
<index> may be an expression.

Special case

If index = 0, SIGMA returns the number of non-zero elements in the array.

12. STR$ (<expr> [,n])

It is proposed to allow an optional second parameter to the STR$ function. This specifies the maximum number of padding-zeros that will be used before switching to e-format. Padding-zeros mean the zeros that are printed after the last significant digit of a whole number to show place value (position of unprinted decimal point), or the number of zeros printed between the decimal point and the first significant digit of a fraction

eg.

```
    STR$(1000000)       = "1000000"
but STR$(1000000,5)     = "1e6"
    STR$(.00001)        = "0.00001"
but STR$(.00001,3)      = "1e-5"
```

13. REF operator

It is proposed to allow the creation of pointer types using the REF operator. The usage proposed is that an assignment of the form:

i#:= REF <numeric variable or array>

- creates a reference value for the nominated variable. i# is either created, or promoted to a reference type, and cannot be subsequently used as a normal integer. Any future use of i# results in immediate de-referencing, so only one level of reference variable is allowed.

Example

```
10    a:=0
20    i#:= REF a
30    i#:+1
40    PRINT a
run
1
END AT 40
```

To transfer one reference variable to another, you must write (line 30)

```
10    a:=0, b:=0
20    i#:=REF a;j#:=REFb
30    i#:=REF j
```

- the statement

```
30    i#:=j#
```

would transfer the value of b to a

- whereas

```
30    i#:=REF j#
```

re-references the de-referenced j# so that i# points to b

Subarrays

It is often desireable to be able to address part of an array in an analogous manner to a substring. It is proposed to utilise the same syntax and similar semantics to a substring. For this purpose, the number of dimensions, and index starting values are ignored; the array is regarded as a 1-dimensional array that starts at element nr 1. Thus a(1:1) is a(first, first, first ...) and a(-1:-1) is a(last, last, last,...). If a is 1-dimensional and starts at element 1, then a(1:n) represents the first n elements of a.

Usage

The syntax may be used in any context in which the array name without subscripts is valid.

E.g. LHS of assignment,
PRINT list item
READ FILE item
WRITE FILE item
INPUT FILE item
PRINT FILE item